Programming in Haskell – Homework Assignment 8

UNIZG FER, 2014/2015

Handed out: December 13, 2014. Due: December 20, 2014 at 23:59

Note: Define each function with the exact name and the type specified. You can (and in most cases you should) define each function using a number of simpler functions. Provide a type signature above each function definition and comment the function above the type signature. Unless said otherwise, a function may not cause runtime errors and must be defined for all of its input values. Use the `error` function for cases in which a function should terminate with an error message. Problems marked with a star ($\star$) are optional.

Each problem is worth a certain number of points. The points are given at the beginning of each problem or subproblem (if they are scored independently). These points are scaled, together with a score for the in-class exercises, if any, to 10. Problems marked with a star ($\star$) are scored on top of the mandatory problems, before scaling. The score is capped at 10, but this allows for a perfect score even with problems remaining unsolved

1. *(1 pt)* Define a small domain-specific language for enslaving turtles. Using the data definitions below, implement the basic functions needed to control the turtle.

```
-- x and y coordinates
type Position = (Integer, Integer)
data Orientation = Left | Right | Up | Down deriving (Eq, Show)
-- Clockwise and Counterclockwise
data TurnDir = CW | CCW deriving (Eq, Show)
```

   (a) Define the data type Turtle in such a way to make solving the other subtasks possible.

   `data Turtle = {...} deriving Show`

   (b) Define the function `position` that takes a `Turtle` and returns its `Position`. (Hint: No need to define it explicitly. Recall record syntax).

   `position :: Turtle -> Position`

   (c) Define the function orientation that takes a Turtle and returns its Orientation. (Hint: Same as above)

   `orientation :: Turtle -> Orientation`

   (d) Define the function `newTurtle` that creates a `Turtle` at Position `(0, 0)` facing upwards.

```
leonardo = newTurtle
position leonardo ⇒ (0,0)
orientation leonardo ⇒ Up
```

(e) Define the function `moven` that takes an `Integer` and a `Turtle` and moves the turtle a given amount in the direction it is currently facing.

```
move :: Integer -> Turtle -> Turtle
position $ move 20 leonardo ⇒ (0,20)
position $ move 0 leonardo ⇒ (0,0)
position $ move (-10) leonardo
⇒ error "Turtles cannot move backwards"
-- The above is not true
```

(f) Define the function `turn` that takes a `TurnDir` and a `Turtle` and changes the turtle's position accordingly.

```
turn :: TurnDir -> Turtle -> Turtle
orientation $ turn CCW leonardo ⇒ Left
orientation $ turn CW $ turn CCW leonardo ⇒ Up
position $ move 5 $ turn CW leonardo ⇒ (5,0)
position $ move 5 $ turn CCW $ move 10 $ turn CCW leonardo
⇒ (-10,5)
```

(g) Implement a function `runTurtle` that enables us to chain our commands to the turtle more easily (Disclaimer: The name `slowWalkTurtle` would have been more apt, but `runTurtle` was chosen for brevity).

```
runTurtle :: [Turtle -> Turtle] -> Turtle -> Turtle
position $ runTurtle [move 5, turn CW, move 20, turn CW, move 10]
leonardo ⇒ (20,-5)

spiral i = move i : turn CW : spiral(i + 1)
position $ runTurtle (take 10 $ spiral 1) newTurtle ⇒ (-2,3)
```

2. *(1 pt)* Implement the following functions over a `Tree` data type defined as:

```
data Tree a = Leaf | Node a (Tree a) (Tree a) deriving (Eq, Show)
testTree = Node 1 (Node 2 Leaf Leaf) (Node 3 (Node 4 Leaf Leaf) Leaf)
```

(a) Define a function `treeFilter` that takes a predicate and a `Tree` and removes those subtrees that do not satisfy the given predicate (with any children).

```
treeFilter :: (a -> Bool) -> Tree a -> Tree a
treeFilter (const True) testTree ⇒ testTree
treeFilter odd testTree ⇒ Node 1 Leaf (Node 3 Leaf Leaf)
treeFilter (<3) testTree ⇒ Node 1 (Node 2 Leaf Leaf) Leaf
```

(b) Define a function `levelMap` that takes some binary function and applies it to the tree. The function that is being applied takes the depth of the tree as the first argument. The root element's depth is 0.

```
levelMap :: (Int -> a -> b) -> Tree a -> Tree b
levelMap (\l x -> if odd l then x else x + 1) testTree
⇒ Node 2 (Node 2 Leaf Leaf) (Node 3 (Node 5 Leaf Leaf) Leaf)
levelMap const testTree)
⇒ Node 0 (Node 1 Leaf Leaf) (Node 1 (Node 2 Leaf Leaf) Leaf)
```

(c) Define a function `isSubtree` that takes two instances of `Tree` and checks whether the first tree appears as part of the second.

```
isSubtree :: Eq a => Tree a -> Tree a -> Bool
isSubtree (Node 4 Leaf Leaf) testTree ⇒ True
```

```
isSubtree (Node 3 Leaf Leaf) testTree ⇒ False
isSubtree testTree testTree ⇒ True
isSubtree Leaf testTree ⇒ True
```

3. *(1 pt)* Define a `Date` type and the implement the following functions for working with dates.

```
data Date = Date { day :: Int
                 , month :: Int
                 , year  :: Int
                 } deriving (Eq, Show)
```

   (a) Define a function `date` that constructs a date from three integers. It is to be used instead of the `Date` constructor to check for date validity. If the date is invalid, `Nothing` is returned. Otherwise, `Just Date` is returned.

```
date :: Int -> Int -> Int -> Maybe Date
date 10 10 2014 ⇒ Just (Date 10 10 2014)
date 31 8 2014 ⇒ Just (Date 31 8 2014)
date 31 9 2014 ⇒ Nothing
date 35 9 2014 ⇒ Nothing
date 29 2 2014 ⇒ Nothing
date 29 2 2012 ⇒ Just (Date 29 2 2012)
date (-1) 0 1000 ⇒ Nothing
date 10 5 (-100) ⇒ Just (Date 10 5 (-100))
```

   (b) Define a function `addDays` that takes a `Date` and an `Int` as a number of days to add and calculates the new `Date`.

```
addDays :: Date -> Int -> Date
addDays (Date 12 8 2014) 3 ⇒ Date 15 8 2014
addDays (Date 30 8 2014) 2 ⇒ Date 1 9 2014
addDays (Date 12 3 2014) (-2) ⇒ Date 10 3 2014
addDays (Date 1 3 2014) (-1) ⇒ Date 28 2 2014
addDays (Date 31 12 2014) 1 ⇒ Date 1 1 2015
```

4. *(1 pt)* Define a recursive data type `Pred` that represents a boolean expression. Use the type constructors `And`, `Or` and `Not` to represent boolean operations and the `Val` constructor to represent a boolean value. Implement the `eval` function that takes a `Pred` and returns its evaluated `Bool` value.

```
expr = And (Or (Val True) (Not (Val True))) (Not (And (Val True)
           (Val False)))

eval (Val True) ⇒ True
eval (Val False) ⇒ False
eval (Or (Val True) (Val False)) ⇒ True
eval expr ⇒ True
eval (Not expr) ⇒ False
```

5. *(2 pts)* Define the following custom data types:

```
data StackTraceElement = StackTraceElement { className  :: String
```

```
                                                      , method     :: String
                                                      , lineNumber :: Int
                                                      }
type StackTrace = [StackTraceElement]
```

A stack trace containing the following information:

```
Main.main:12
Mapper.map:44
Decoder.prepare:4
Decoder.decode:234
```

would be represented as follows in this model:

```
ste11 = StackTraceElement
          { className = "Main"
          , method="main"
          , lineNumber=12 }
ste12 = StackTraceElement
          { className = "Mapper"
          , method="map"
          , lineNumber=44 }
ste13 = StackTraceElement
          { className = "Decoder"
          , method="prepare"
          , lineNumber=4 }
ste14 = StackTraceElement
          { className = "Decoder"
          , method="decode"
          , lineNumber=234 }
st1 = [ste11, ste12, ste13, ste14]
```

NB: In `StackTrace` a succesor element is a child of the previous element. (Order matters.)

Write a function `combined` that produces a `String` containing information of all stack traces combined. Output is organized in a tree–like form with combined parents and childern ordered alphabetically. For every child element you must use intendation by `k`, where `k` is argument provided to function.

```
ste21 = StackTraceElement
          { className = "Main"
          , method="main"
          , lineNumber=12 }
ste22 = StackTraceElement
          { className = "Mapper"
          , method="map"
          , lineNumber=44 }
ste23 = StackTraceElement
```

```
          { className = "Decoder"
          , method="order"
          , lineNumber=7 }
st2 = [ste21, ste22, ste23]

ste3 = StackTraceElement
         { className = "Main"
         , method="run"
         , lineNumber=33 }
st3 = [ste3]

combined :: [StackTrace] -> Int -> String
combined [st1,st2,st3] 4 =>
2 Main.main:12
    2 Mapper.map:44
        1 Decoder.order:7
        1 Decoder.prepare:4
            1 Decoder.decode:234
1 Main.run:33
```

You can use any custom data types and helper functions as long as you write them yourself. Watch out for corner cases.

6. *(1 pt)* Reflected binary code, also known as Gray code, is a binary system where successive numbers differ by only one bit. That feature makes them useful in error correction, genetic algorithms, etc. These subproblems are to be solved using the `Data.Bits` module. The module defines `Int` and `Integer` as instances of a new `Data.Bits` typeclass, so we will limit our function to `Integral` representations of bits. (Hint: Check the `xor` and `shiftR` functions.)

   (a) Implement the function `toGrayCode` that takes a number in natural binary representation and returns it in Gray code.

   ```
   toGrayCode :: (Integral a, Bits a) => a -> a
   toGrayCode 0 ⇒ 0
   toGrayCode 1 ⇒ 1
   toGrayCode 3 ⇒ 2
   toGrayCode 10 ⇒ 15
   ```

   (b) Implement the function `fromGrayCode` that takes a number in Gray code and returns its natural binary representation.

   ```
   fromGrayCode :: (Integral a, Bits a) => a -> a
   fromGrayCode 0 ⇒ 0
   fromGrayCode 3 ⇒ 2
   fromGrayCode 15 ⇒ 10
   fromGrayCode 103 ⇒ 69
   ```

7. *(1 pt)*

   (a) Define a typeclass `Truthy` that defines types that can be interpreted as boolean values. The typeclass must offer two functions:

```
truey :: a -> Bool
falsey :: a -> Bool
```

Provide default implementations so that the minimal definition for the `Truthy` typeclass requires the definition of only one of those functions (either). Make `Bool`, `Int` and `[a]` instances of `Truthy`. For `Int`, only 0 should be considered falsey, while for lists only an empty list should be considered falsey.

(b) Define a function `if'` that works on instances of `Truthy` and behaves like the *if-then-else* construct.

```
if' :: Truthy p => p -> a -> a -> a
if' [1,2,3] "True" "False" ⇒ True
if' (1 > 2) "GT" "LE" ⇒ "LE"
if' 0 1 2 ⇒ 2
```

(c) Define a function `assert` that takes a `Truthy` value and another argument. If the first argument evaluates to truey, it returns the second argument. Otherwise it raises an error.

```
assert :: Truthy p => p -> a -> a
assert [1,2,3] [4,5,6] ⇒ [4,5,6]
assert [] (-2) ⇒ error "Assertion failed"
```

(d) Define the `(&&&)` and `(|||)` functions that behave like the `(&&)` and `(||)` functions, but operate on `Truthy` instances instead of `Bool`.

```
(&&&) :: (Truthy a, Truthy b) => a -> b -> Bool
(|||) :: (Truthy a, Truthy b) => a -> b -> Bool

[1] &&& [2] ⇒ True
1 ||| 0 ⇒ True
[1,2] &&& 0 ⇒ False
[1,2] ||| 0 ⇒ True
```

8. *(1 + 1⋆ pt(s))* We know that the `(++)` concatenation operator can be expensive, especially when repeated multiple times (e.g., in pretty-printing). For that reason, you shall implement a difference list. A difference list stores the concatenation as a computation. For that reason, all appending operations are `O(1)` as long as we are working with difference lists – the only time we truly need to perform the concatenation is when we are transforming the difference list into a list.

   (a) Implement a datatype `DiffList` that holds a concatenation computation - given a list it will hold a `function` - the concatenation of that list and another list it expects as an argument. Use record syntax to define a function `undiff` that takes a list and returns the concatenation of the `DiffList` and that list (as a list).

   ```
   data DiffList a = { undiff :: ... }
   -- assuming dl is a DiffList containing the computation [1,2]++[3]++x:
   undiff dl [4] ⇒ [1,2,3,4]
   ```

   (b) Implement a function `empty` that constructs an empty `DiffList`.

   ```
   empty :: DiffList a
   undiff empty [] ⇒ []
   undiff empty [2,3] ⇒ [2,3]
   ```

(c) Implement a function `fromList` that takes a list and returns a `DiffList` as its concatenation computation.

```
fromList ::  [a] -> DiffList a
undiff (fromList [1,2,3]) [4,5]  ⇒  [1,2,3,4,5]
```

(d) Implement a function `toList` that takes a `DiffList` and returns its computation, a concatenated list.

```
toList ::  DiffList a -> [a]
toList $ fromList xs == xs
```

(e) Implement a function `append` that takes two `DiffLists` and combines them into a new `DiffList`. (Hint: Note that this amounts to function composition since contents of `DiffLists` are functions)

```
append ::  DiffList a -> DiffList a -> DiffList a
toList $ append dl dl  ⇒  [1,2,3,1,2,3]
```

(f) *(1⋆ pt)* Declare `DiffList` an instance of Monoid. Monoids have to have an associative binary operation and an identity element – that means that for the operation `op` and an identity element `null` it will hold that `x op null == null op x == x` for any `x`. A monoid also has to satisfy some laws (known as the `monoid laws`) that are derived from this. The minimal complete definition for a `Monoid` consists of two functions – `mempty`, that defines the identity element, and `mappend`, that defines the associative binary operator.

9. *(2 pts)* The floating point number representation only approximates real numbers. We can demonstrate this by evaluating a common example:

```
ghci> sum $ replicate 10 0.1 :: Float
1.0000001
ghci> sum $ replicate 10 0.1 :: Double
0.9999999999999999
```

There are cases where we wish to avoid these inaccuracies, and for this purpose we will need our own exact rational number type and its associated operations. Create a module named `Ratio` (placing it in `Ratio.hs`) and within it define the following:

(a) The type `Ratio`, representing a rational number as a ratio of two `Integers`. How you define the constructor(s) is up to you. Make sure the module only exports the *type* `Ratio`, and *not* its constructor(s). We wish to forbid the users of this library from tinkering with the low-level representation of our type by calling the constructor(s) directly.

(b) The operator `(%)`, returning a `Ratio` given two values convertible to `Integer`. The users of this library will create `Ratios` using this operator, so be sure to export it. Use whatever precedence and fixity makes sense.

```
(%) :: (Integral a, Integral b) => a -> b -> Ratio
(five, three, tenth) = (5 % 1, 15 % 5, 1 % 10)
2 % 0 = error "Division by zero"
```

(c) The instance `Eq Ratio`, allowing us to compare whether two `Ratios` are equal. Hint: you can see all of the `Eq` member functions by running ":info Eq" within ghci, and similarly for any other class.

```
ghci> five /= three
True
```

```
ghci> three == (3 % 1)
True
```

(d) The instance `Ord Ratio`, allowing us to compare whether a `Ratio` is greater or lesser than another. Hint: you don't have to implement all the member functions. For instance, simply implementing `(<=)` is enough, since all the other operators can be expressed using it (and the `Eq` instance).

```
ghci> five > three
True
ghci> tenth `compare` five
LT
```

(e) The instance `Num Ratio`, defining all of the seven `Num` member functions. This allows us to perform very basic arithmetic with our new type.

```
ghci> sum (replicate 10 tenth) == (1 % 1)
True
```

Additionally, once you've implemented the `fromInteger` function, GHC learns how to create a `Ratio` from an `Integer`. This allows you to start defining the simplest ratios by omitting the denominator. Note `ten` and the literal `3` in the following example:

```
ghci> let ten = 10 :: Ratio
3 * 10 == (90 % 3)
True
```

(f) And finally, the instance `Show Ratio`. Make sure to show the simplest possible form. Hint: you can simplify a ratio with the help of `Prelude.gcd`, which returns the greatest common divisor of two numbers. This allows you to determine, for instance, that `(14 % 4)` can be represented more simply by `(7 % 2)`.

```
ghci> tenth
1 % 10
ghci> (five, three)
(5 % 1, 3 % 1)
```

Corrections

Revision 1.1 – Corrected `type -> data` for `Orientation` and `TurnDir`. Added parameter `a` for `Tree`. Changed `even` to `odd` in one example of `levelMap`. Added some parantheses around negative numbers in Problem 3. Changed `Val` to `eval` in Problem 4. Thank you Luka Horvat, you saved the village!
Revision 1.2 – Corrected the `spiral` example for `runTurtle`.
Revision 1.3 – Corrected `odd` example for `Tree`. Again. Added some types and a subtask for `DiffList`. Fixed a `addDays` example ($-1 \to 1$). Added some type signatures we didn't remember to add. Added some `Val` constructors I accidentally deleted in Revision 1.1
Revision 1.4 – Added points (or, actually, point) for Problem 4.